CS3025 Compiladores

Laboratorio 4.2 – 8 Septiembre 2023

En el laboratorio anterior exploramos la implementación de *parsers* de descenso recursivo y extensiones para asignar acciones al análisis sintáctico. Logramos implementar lo siguiente:

- Un *parser* de expresiones aritméticas que, además de realizar el análisis sintáctico, evaluaba el input. En pocas palabras, implementamos una calculadora.
- Un *parser* de un pequeño lenguaje imperativo que, además de realizar análisis sintactico, ejecutaba el input. En pocas palabras, se implementó un Intérprete.

Tambien observamos que, al incrementarse la complejidad de la gramática de nuestro lenguaje, la estrategia de mezclar análisis sintactico con acciones semánticas para evaluación e interpretación no era escalable. Para obtener escabilidad es necesaria una estrategia diferente: construir un árbol sintactico abstracto (AST: Abstract Syntax Tree)

Archivos necesarios: `exp.hh`, `exp.cpp`, `exp_test.cpp`, `aexp_parser.hh` y `aexp_parser.cpp`.

1. Analisis Sintactico y construcción de expresiones aritmeticas

En esta primera parte implementaremos un *parser* que, en lugar de evaluar expresiones aritmeticas, construye ASTs.

Empecemos con los ASTs. Los archivos exp.hh y exp.cpp definen e implementan la estructura y funcionalidad de los arboles que representaran nuestras expresiones aritmeticas. La clase abstracta Exp define la interface:

```
class Exp {
public:
  virtual void print() = 0;
  virtual int eval() = 0;
  static string binopToString(BinaryOp op);
  virtual ~Exp() = 0;
};
```

Ademas tenemos 3 clases para representar expresiones con operaciones binaria, expresiones entre paréntesis y números:

```
enum BinaryOp { PLUS, MINUS, MULT, DIV, EXP};
class BinaryExp : public Exp {
public:
  Exp *left, *right;
  BinaryOp op;
  BinaryExp(Exp* l, Exp* r, BinaryOp op);
… };
```

```
class NumberExp : public Exp {
public:
  int value;
  NumberExp(int v);
… };

class ParenthExp : public Exp {
public:
  Exp *e;
  ParenthExp(Exp *e);
… };
```

Cada clase implementa métodos para imprimir y evaluar: toda la funcionalidad para implementar la calculadora se ha movido a una clase separada.

El archivo `aexp_test.cpp` muestra cómo construir expresiones aritméticas utilizando constructores. El siguiente código construye el AST de 2**3:

```
exp = new BinaryExp( new NumberExp(2), new NumberExp(3), EXP);
```

Compilar y ejecutar de la siguiente manera:

```
>> g++ exp_test.cpp aexp_parser.cpp exp.cpp
>> ./a.out       // sin argumentos
```

Tarea1: Construir expresiones aritméticas más complejas para verificar que todo está en orden e.g. (5 + 3**2) / 2.

Los archivos `aexp_parser.hh` y `parser_aexp_parser.cpp` contienen un analizador sintáctico de expresiones aritméticas regidas por la siguiente gramática[1]:

```
Exp ::= Term ((´+´ | ´-´) Term)*     Term ::= FExp ((´*´|´/´) FExp)*
FExp ::= Factor [** FExp]            Factor ::= num | ´(´ Exp ´)´
```

El análisis sintáctico define una función por cada no-terminal: `parseExpression`, `parseTerm`, `parseFExp` y `parseFactor`. La función de entrada a la gramática llama a `parseExpression` para empezar el análisis.

El objetivo de esta parte del laboratorio es agregar llamadas a constructores al código que implementa el analisis sintactico. Las declaraciones de los métodos reflejan esto:

```
class Parser {
private:
…
  Exp* parseExpression();
  Exp* parseTerm();
  Exp* parseFExp();
  Exp* parseFactor();
```

[1] Usamos ::= en lugar de → para especificar reglas o producciones.

```
public:
  Parser(Scanner* scanner);
  Exp* parse();
};
```

Vemos que ahora los métodos que implementan el descenso recursivo retornan `Exp*`. Modificar el código de `parseFactor` a:

```
Exp* Parser::parseFactor() {
  if (match(Token::NUM)) {
    return new NumberExp(stoi(previous->lexema));
  }
  if (match(Token::LPAREN)) {
    Exp* e = parseExpression();
    if (!match(Token::RPAREN)) {
      cout << "Expecting right parenthesis" << endl;
      exit(0);
    }
    // return e; is fine too
    return new ParenthExp(e);
  }
  cout << "Couldn't find match for token: " << current << endl;
  exit(0);
}
```

Tarea 2: Modificar el resto de los métodos para que retornen punteros a objectos Exp (expresiones aritméticas) insertando los constructores correspondientes. Probar el nuevo código con el programa exp_test.cpp (cambiar el flag `useparser` a `true`)

2. Otro interprete para IMP

En el laboratorio pasado vimos el lenguaje imperativo IMP con la siguiente gramática

```
Program ::= StmtList
StmtList := Stmt ( ';' Stmt )*
Stmt ::= id '=' Exp  | 'print' '(' Exp ')'
Exp ::= Term (('+' | '-') Term)*    Term ::= Factor (('*'|'/') Factor)*
FExp ::= Factor [** FExp]  Factor ::= id | num | '(' Exp ')'
```

El AST para programas IMP esta implementado por `imp.hh` e `imp.cpp`. Las nuevas definiciones (estructura y constructores) son:

```
class Stm {
public:
  virtual void print()=0;
  virtual void execute()=0;
  virtual ~Stm() = 0;
};

class AssignStatement : public Stm {
private:
  string id; Exp* rhs;
```

```
public:
  AssignStatement(string id, Exp* e);
  …
};

class PrintStatement : public Stm {
private:
  Exp* e;
public:
  PrintStatement(Exp* e);
…
};

class Program {
private:
  static unordered_map<string, int> memoria;
  list<Stm*> slist;
public:
  Program();
  void add(Stm* s);
… }
```

El metodo `execute()` ejecuta el programa e instrucciones usando la variable estatica `memoria`. La funcionalidad es la misma que cuando teníamos el interprete junto al *parser* pero, esta vez, la tenemos en clases dedicadas: El AST.

El programa `imp_test.cpp` muestra como podemos construir un pequeño programa IMP usando constructores:

```
Exp* e = new BinaryExp( new NumberExp(2),
                        new NumberExp(3), EXP);
Stm* s1 = new AssignStatement("x",e);
Stm* s2 = new PrintStatement(new IdExp("x") );
program->add(s1);
program->add(s2);
```

Tarea 3: Modificar el parser para que produzca los ASTs

Los archivos `imp_parser.hh` e `imp_parser.cpp` contienen las declaraciones necesarias para implementar la construcción de ASTs: los métodos del parser retornan Exp*, Stm* o Program*. Agregar código para que llame a los constructores respectivos.

3. Bison y Flex

Hasta ahora hemos implementado todos nuestros parsers “a mano”. La semana pasada mostramos como generar scanners de manera automática usando flex. Del mismo modo, la herramienta Bison genera parser de manera automática.

Tarea: Instalar Bison y probar que efectivamente puede generar parsers con código en C++.

Para esto, necesitamos primero bajar los archivos lexer1.l y grammar1.y. Ejecutar los siguientes comandos:

```
>> flex lexer1.l
>> bison grammar1.l
>> g++ -o calc1 Scanner1.cpp Parser1.cpp
>> ./calc1
```

El programa calc1 implementa una calculadora que lee expresiones de standard input.